A Storage Structure for Nested Relational Databases

by

Anand Deshpande and Dirk Van Gucht
Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 234

A Storage Structure for Nested Relational Databases
by Anand Deshpande and Dirk Van Gucht

November, 1987

This report to appear in *Nested Relations and Complex Objects,* S. Abiteboul and H.-J Schek, (Eds.), Springer-Verlag, 1988.

A Storage Structure for Nested Relational Databases

Anand Deshpande

Dirk Van Gucht

Computer Science Department
Indiana University
Bloomington, IN 47405, USA

We propose a storage structure for Nested Relational Databases. In particular, we define a combination of two data structures: a record-list structure and a tree storing all the atomic values present in the tuples and sub-tuples of the database. This storage structure allows for efficient execution of updates and queries expressed in the extended relational algebra of the nested relational database model.

1. Introduction

In 1977 Makinouchi [16] proposed to generalize the relational model by removing the first normal form assumption. Jaeschke and Schek [12] introduced a generalization of the ordinary relational model by allowing relations with set-valued attributes and adding two restructuring operators, the nest and the unnest operators, to manipulate such (one-level) nested relations. Thomas and Fischer [26] generalized Jaeschke and Schek's model and allowed nested relations of arbitrary (but fixed) depth. Roth, Korth and Silberschatz [24] defined a calculus like query language for the nested relational database model (NRDM) of Thomas and Fischer. Since then numerous SQL-like query languages [15, 21, 22, 23], graphics-oriented query languages [11] and datalog-like languages [2, 3, 5, 14] have been introduced for this model or slight generalizations of it. Also, various groups [4, 7, 8, 9, 19, 25] have started with the implementation of the nested relational database model, some on top of an existing database management system, others from scratch.

The purpose of this paper is to address some issues related to the implementation of the NRDM by defining a storage structure for nested relational databases. In particular, we define a combination of two data structures:

- `VALTREE` – a tree structure storing all the atomic values present in the tuples and sub-tuples of the database, and
- `RECLIST` – record-list structures which store data as tuples and sub-tuples.

The `VALTREE` is a generalization of the domain based approach proposed by Missikoff [17] and Missikoff and Scholl [18] for the relational model. The `RECLIST` is inspired by some of the storage structures proposed by Dadam et.al. and Deppisch et.al. [7, 8]. We will argue that the proposed storage structure allows for the efficient execution of updates and queries.

In Section 2 we discuss the nested relational model and describe an algebra for this model. In Section 3 we describe a notation for tuple identification and then discuss `VALTREE` and `RECLIST` storage structures. In Section 4 we discuss how these storage structures could be used to efficiently execute nested algebra queries. Finally, in Section 5 we discuss important observations about this implementation and also discuss issues that need further investigation.

2. The Nested Relational Database Model

In this section we briefly describe the nested relational database model and the algebra associated with it. In this model the database is defined as a set of nested relational structures, i.e., a component of a tuple in such a structure can be an atomic value or a nested relational structure. This is in contrast with the classical relational model where a database is defined as a set of flat relations in which components of the tuple are always atomic values.

The class of nested structures we have restricted ourselves to in our implementation is referred to as the class of hierarchical structures [1,6,27]. The difference between hierarchical structures and general nested structures is that in a hierarchical structure a combination of atomic attributes form the key at each level of the structure, whereas this restriction is not imposed on a general nested structure.

The algebra used for manipulating hierarchical structures is an extension of mechanisms used to manipulate flat relations, with the addition of some restructuring operators. The algebraic operators that are used for the implementation are described informally as follows (for more detail see [1,6,27]):

Union ($\bigcup^e$) : The union operator has to be extended to ensure that the resulting structure is hierarchical, i.e. the key property is maintained. The extended union operator involves a union of all tuples and then a fusion of set components with identical atomic values.

Difference ($-^e$) : The difference operator is extended in a similar manner as the union operator.

Project (π^e) : The standard project operator does not maintain hierarchical structures. This can be observed in Example 4. Therefore project, like union has a fusion component associated with sets components that have identical key values.

Select (σ^e_C) : The select operator though very similar to the relational model is far more expressive in this model. Selects can be performed at any level of the structure. For convenience a template for filtering tuples is provided [6]. Besides standard conditional operators ($=, \neq, >, \geq, <, \leq$) which can be used for comparing atomic values we also need ($\in, \notin, \subset, \not\subset, \subseteq, \not\subseteq$) to compare sets and elements of the set.

Join ($\bowtie$) : At the present moment we discuss implementation of natural joins when the pivot attributes in the two structures to be joined have been restricted to atomic values only. An example of the join operator is shown in Figure 6. It would be interesting to study how joins over set-valued attributes should be implemented. This problem is non-trivial because it is difficult to index on sets of values. Conditional joins are also not discussed in this paper.

Nest (ν) : The nest operator we consider in our implementation, is restricted in such a way that the result of a nest operation yields a structure with only one set-valued attribute at the highest level. To construct structures with multiple set-valued attributes one needs to use a combination of nest and join operations. Figure 4 shows an example of a nest operation.

Unnest (μ) : The unnest operator is the inverse of the nest operator. An example of the unnest operator is shown in Figure 3. Again at the moment we restrict ourselves to unnesting one level at a time.

Example 1: Let us consider an example with two nested relational structures, SCHEDULE and AIRLINE-INFO, as shown in Figure 1 and Figure 2. The structure SCHEDULE stores information about universities their nearest airports and a list of their away football games while the AIRLINE-INFO structure stores information about cities, flights departing the city and airlines for which the city is a hub. These structures have been chosen to illustrate some typical features of our approach and will be used as examples throughout this paper.

TEAM	NEAREST-AIRPORTS	TEAMS-TO-PLAY
Indiana	Indianapolis Cincinnati Louisville	Purdue Michigan Wisconsin
Purdue	Indianapolis Chicago	Minnesota Iowa Michigan
Northwestern	Chicago	Ohio State Iowa Minnesota
Michigan	Detroit	Micigan State Ohio State Wisconsin
Michigan State	Detroit	Indiana Purdue Iowa
Illinois	Chicago St. Louis	Indiana Ohio State Northwestern

Figure 1: The SCHEDULE structure

The following examples illustrates some operators of the algebraic query language for the NRDM.

Example 2: $\text{SCHEDULE}' = \mu_{\text{NEAREST-AIRPORTS}}\pi^{e}_{\text{TEAM,NEAREST-AIRPORTS}}(\text{SCHEDULE})$

This example projects the teams and the nearest airports and then unnests it as shown in Figure 3.

Example 3: $\text{SCHEDULE}'' = \nu_{\text{TEAM}}(\text{SCHEDULE}')$

This example nests the SCHEDULE$'$ structure from the previous example and nests the TEAM component as shown in Figure 4. The resulting structure SCHEDULE$''$ lists cities and the teams that are close to them.

	FLIGHTS		
CITY	DESTINATION	AIRLINES	AIRLINE-HUBS
Indianapolis	Chicago	Transworld United	Transworld United
	NewYork	United Eastern	
	St. Louis	Transworld Continental	
St. Louis	Chicago	Transworld United	Transworld Eastern
	New York	Transworld Eastern	
	Indianapolis	Transworld Continental	
	Detroit	Northwest Transworld	
Chicago	Indianapolis	United Eastern Northwest	United Eastern
	New York	Transworld Northwest United	
	St. Louis	Transworld	
	Detroit	Northwest	
	Los Angeles	United	
New York	Indianapolis	Transworld Eastern	Transworld United Eastern Delta
	St. Louis	Transworld	
	Detroit	Northwest	
	Cincinnati	Delta Eastern	
	Atlanta	Delta Eastern	

Figure 2: The AIRLINE-INFO structure

Example 4: $\texttt{FLIGHTS}' = \pi^{e}_{\texttt{FLIGHTS}}\ (\texttt{AIRLINE-INFO})$

This example projects the structure FLIGHTS from the structure AIRLINE-INFO as shown in Figure 5. Notice how the project operator causes the sets of AIRLINES corresponding to a CITY to merge, e.g observe the AIRLINES set for 'Indianapolis'.

Example 5: $\texttt{TEAM-AIRLINE} = \texttt{SCHEDULE}'' \bowtie \texttt{FLIGHTS}'$

TEAM	NEAREST-AIRPORT
Indiana	Indianapolis
Indiana	Cincinnati
Indiana	Louisville
Purdue	Indianapolis
Purdue	Chicago
Northwestern	Chicago
Michigan	Detroit
Michigan State	Detroit
Illinois	Chicago
Illinois	St. Louis

Figure 3: $\texttt{SCHEDULE}' = \mu_{\texttt{NEAREST-AIRPORT}}\pi^{e}_{\texttt{TEAM,NEAREST-AIRPORT}}\ (\texttt{SCHEDULE})$

TEAMS	NEAREST-AIRPORT
Indiana Purdue	Indianapolis
Purdue Northwestern Illinois	Chicago
Indiana	Cincinnati
Michigan Michigan State	Detroit
Indiana	Louisville
Illinois	St. Louis

Figure 4: $\texttt{SCHEDULE}'' = \nu_{\texttt{TEAM}}(\texttt{SCHEDULE}')$

In this example we join the structure SCHEDULE'' with the FLIGHTS' structure as shown in Figure 6. In our algebra, joins ($\bowtie$) are only defined when the pivot attributes are atomic and the domains are compatible. In this example a join is performed on the NEAREST-AIRPORT attribute of SCHEDULE'' structure and the DESTINATION attribute of the FLIGHTS' structure.

3. A Storage Structure for the NRDM

Nested relational databases cannot be trivially mapped to existing database implementations for the following reasons:

- Tuple components are not necessarily atomic, making the mapping to the relational model difficult.
- Query optimizations that exploit the nested relational model cannot be used when the underlying storage structure is relational.
- Retrievals are often made on components deeply nested within tuples.

DESTINATION	AIRLINES
Chicago	Transworld United
NewYork	United Eastern Transworld Northwest
St. Louis	Transworld Continental
Indianapolis	Transworld Continental United Eastern Northwest
Detroit	Northwest Transworld
Los Angeles	United
Cincinnati	Delta Eastern
Atlanta	Delta Eastern

Figure 5: FLIGHTS$'$ $= \pi^{e}_{\text{FLIGHTS}}$ (AIRLINE-INFO)

TEAMS	NEAREST-AIRPORT	AIRLINES
Indiana Purdue	Indianapolis	Transworld Continental United Eastern Northwest
Purdue Northwestern Illinois	Chicago	Transworld United
Indiana	Cincinnati	Delta Eastern
Michigan Michigan State	Detroit	Northwest Transworld
Illinois	St. Louis	Transworld Continental

Figure 6: TEAM-AIRLINE $=$ SCHEDULE$''$ $\bowtie$ FLIGHTS$'$

- Hierarchical and network models were not developed with high level non-procedural languages in mind.

Several attempts at implementing the nested relational model directly are being made, notable are among them are:

- the AIM project at IBM-Heidelberg [7,21,22],

- DASDBS at TH-Darmstadt [8, 19, 20, 25], and
- the VERSO project at INRIA [4, 6]

Our motivation for proposing two tightly coupled data-structures comes from the observation that in the NRDM, nested algebra operations like select, join and nest are value-driven while project and unnest are not. To implement the value-driven operations it is crucial to be able to efficiently determine which attribute and tuples contain a particular 'value'. In contrast, for structure-oriented operations like project and unnest it is required to efficiently access tuples and their components irrespective of the values contained in them. Data-structures that are well suited for project and unnest are unfortunately not always suitable for the value-driven operations, hence our proposal for two storage structures where one supports value-driven requests effectively, while the other supports structure-oriented operations.

In addition, primary storage on computers has become fairly inexpensive while a disk access is still considerably more expensive than a memory access. We exploit this availability of main memory and indicate methods to cache tuple-identifiers in order to perform queries more efficiently.

3.1. A Notation for Tuple and Component Identification

In the nested relational model queries and updates can be performed on values that are deeply nested. For example, in the structure AIRLINE-INFO of Figure 2 we could select all cities that have flights by 'Northwest'. In this case, selections are to be performed on the AIRLINE attribute which is nested within the FLIGHTS attribute. To efficiently handle this request, it is important for tuple-identifiers at the sub-tuple level to be logically related to the tuple-identifiers of their super-tuples. Also, as some of the components of the tuple could be sets, which in turn could have sets as their components, tuple identifiers cannot be flat but must be hierarchical.

In this section, we introduce a notation for identifying tuples and their components. Let the database consist of a finite set of structures $\{r, s, t, ...\}$. The notation for the identification of tuples and their components uses these relation names tagged with subscripts and superscripts. The subscripts take us down the tuples and the superscripts take us across the components.

Example 6: We will illustrate our notation on the (CITY (DESTINATION AIRLINES*)* AIRLINE-HUB*)* structure of Example 1. Thus, for structure t corresponding to the AIRLINE-INFO structure of Figure 2 the tuples would be identified as t_1, t_2, t_3 and t_4 Each tuple is made up of three components: a CITY component, a FLIGHT component and AIRLINE-HUB component. Thus, the first tuple t_1 has three components ${t_1}^a$, ${t_1}^b$ and ${t_1}^c$, where ${t_1}^a$ corresponds to the CITY component, ${t_1}^b$ corresponds to the FLIGHTS component and ${t_1}^c$ corresponds to the AIRLINE-HUB component. Each of these components is either an atomic value or a structure. In our example ${t_1}^a$ is an atomic value, whereas ${t_1}^b$ and ${t_1}^c$ are structures. The structures ${t_1}^b$ and ${t_1}^c$ consists of sub-tuples, so we need to descend one level. The tuples of the structure ${t_1}^b$ are identified as ${t_1}^b{}_1$, ${t_1}^b{}_2$ and ${t_1}^b{}_3$.

The identifiers for the components of the tuple $t_1{}^b{}_1$ are $t_1{}^b{}_1{}^a$ and $t_1{}^b{}_1{}^b$ corresponding to the DESTINATION and AIRLINES components.

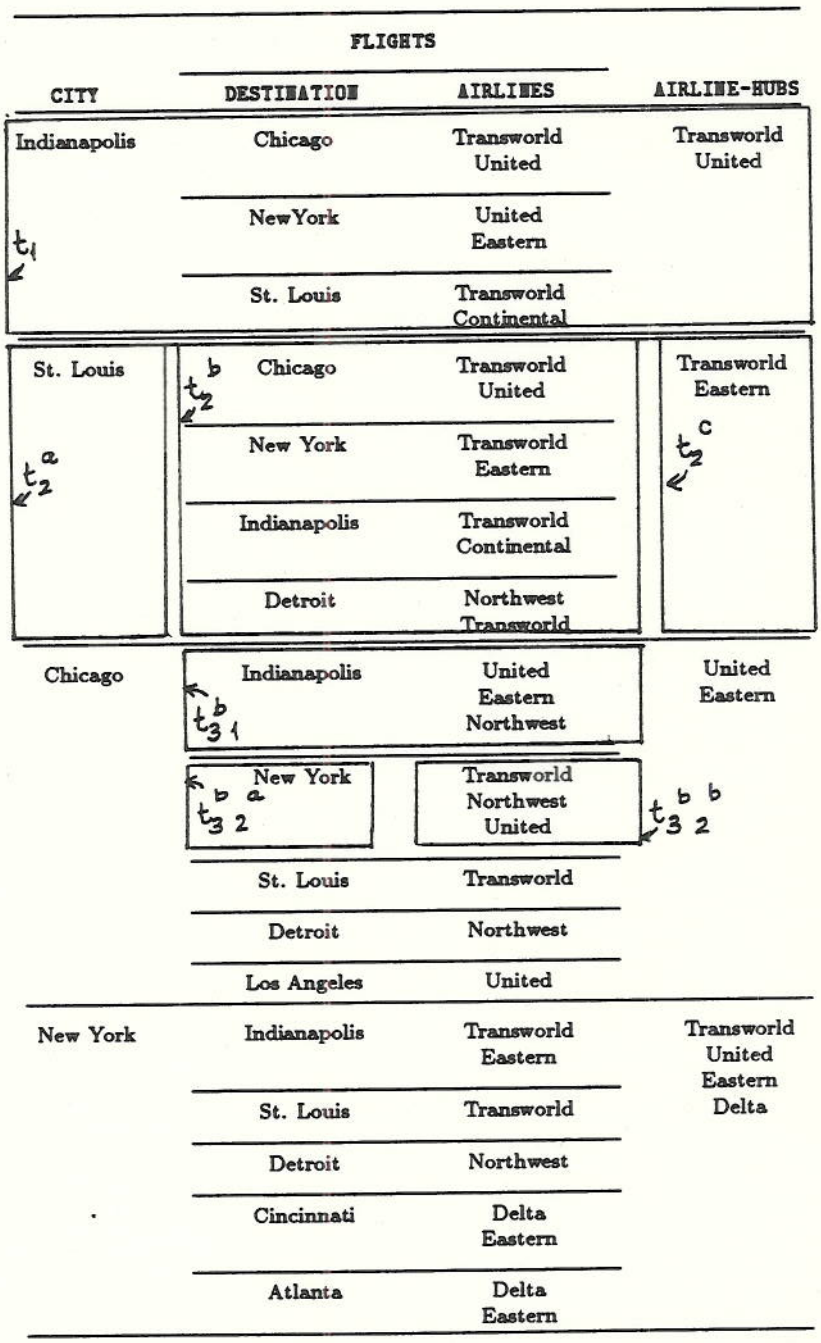

Figure 7: Tuple Notation for AIRLINE-INFO structure

In Figure 7, the notation is illustrated on the structure of Example 1. An interesting feature of this notation is that once we get a tuple or component identifier, we can

trace which tuples or sub-tuples the tuple or component identifier belongs to by going through the superscript strings and the subscript strings.

The length of the subscript and superscript strings depends on the depth of the hierarchical schema of the database. As the schema of the database is fixed, the lengths of the subscript and superscript strings are also fixed.

3.2. The Value-Driven Indexing Scheme

Traditional relational database management systems use indexing techniques to improve access time. Typically, indexes are built on all or some of the attributes of a relation. A value of the index maps to a list of tuple-identifiers of tuples that contain the value of the indexed attribute. Our approach to indexing follows the domain based approach suggested by Missikoff for relational databases [17]. In Missikoff's approach, an atomic value maps to a list of tuple identifiers of tuples in all relations in the database which contain that value. We generalize this approach storing a mapping from a value to a list of all tuple identifiers of tuples in all structures and sub-structures in the nested relational database which contain that value in the `VALTREE`. Hence, given an atomic value, the `VALTREE` returns a set of hierarchical tuple-identifiers, which enables us to determine directly which tuples or sub-tuples the value is stored in. Unlike the conventional database scheme where we have a separate tree for each indexed attribute, our scheme has only one tree, denoted `VALTREE`, that spans over all the atomic values of the database.

We now describe the `VALTREE` in more detail. `VALTREE` is made up of five different levels. The top-most level is called the `DOMAIN` level. This level separates the non-compatible domains into separate sub-trees. The second level, the `VALUE` level, stores all the atomic values of the database. The third level is the `ATTRIBUTE` level. At this level, we store all the attributes that a particular value of the `VALUE` level belongs to. As the same attribute may belong to more than one structure, we have the fourth level called the `STRUCTURE` level. Finally, the fifth and the lowest level consists of all the tuple-identifiers (tid) that correspond to the the atomic value stored at the `VALUE` level; this level is called the `IDENTIFIER` level. The advantage of using the `VALTREE` is that given a value it provides us rapid access to the list of tuple-identifiers corresponding to all occurrences of the value throughout the entire database.

The following observations can be made regarding the `VALTREE` structure:

1. An indexing technique like a B+Tree would be an appropriate data-structure at the `VALUE` level.
2. The granularity of the domain level depends on the database administrator and may depend on the installation. Typically, all compatible attributes belong to the same domain.
3. Values for attributes which do not participate actively in value-driven queries (i.e. remarks fields) need not be stored in the `VALTREE`.
4. It is possible to merge the attribute level and structure level into one if we avoid conflicts in attribute names over structures.

Example 7: In Figure 8, we show parts of the VALTREE for the structures t corresponding to the AIRLINE-INFO structure and s corresponding to the SCHEDULE structure of Example 1.

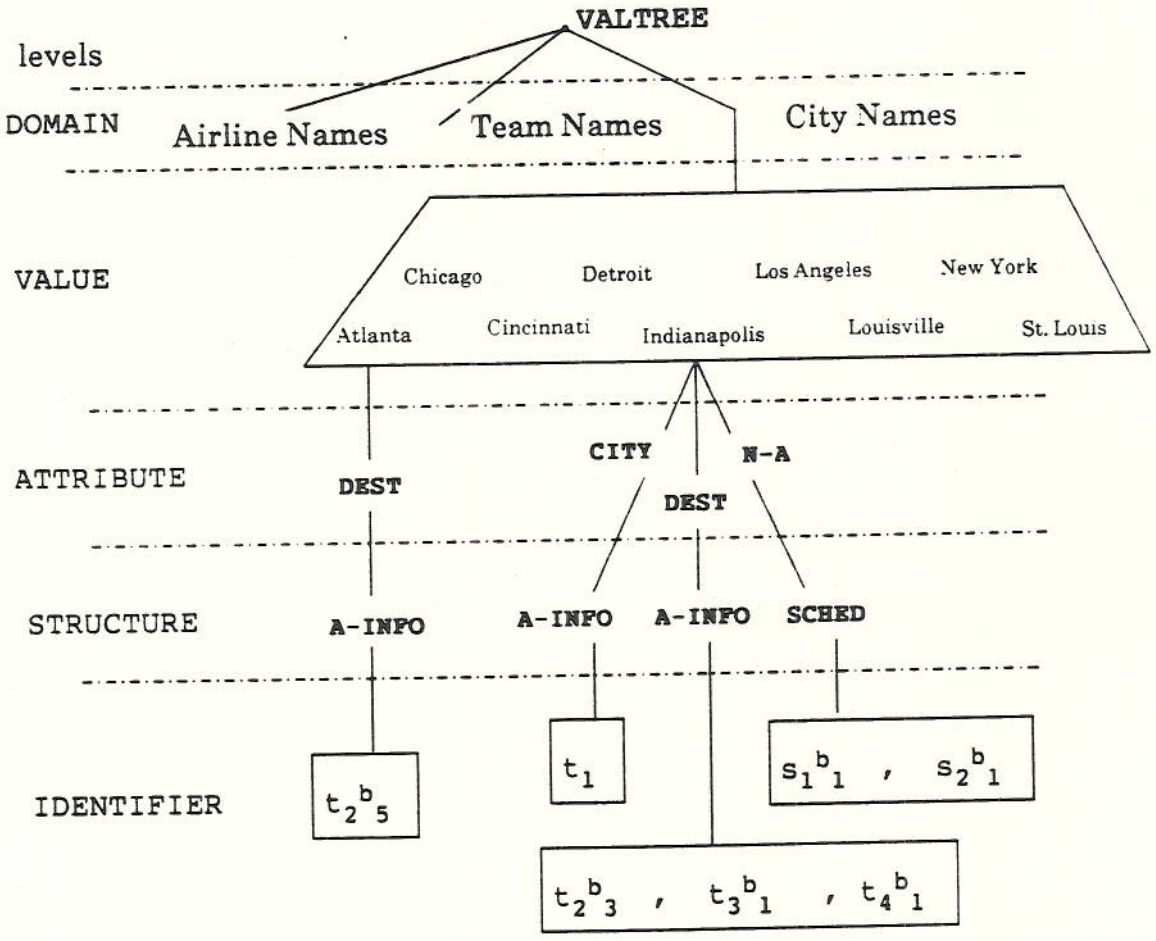

Figure 8: The VALTREE structure

3.3. The Record-List Structure

As the VALTREE is a suitable data structure for performing value-driven operations RECLIST structures have the following requirements:

- Each structure in the nested relational database has a separate RECLIST structure.
- Given a tuple-identifier and an attribute, the number of disk accesses to access the component of the tuple associated with the tuple-identifier and attribute should be minimal.
- The number of disk accesses to retrieve an entire tuple should be minimal; therefore tuples should be clustered as closely as possible.

- It should be easy to do structure-oriented operations
- It should be possible to traverse through the entire structure a tuple at a time.

To achieve these goals we compare several schemes and discuss their trade-offs.

3.3.1. Simple Pointer Structure

The simple pointer scheme consists of storing the tuples and sub-tuples as a linked list. As shown in the Figure 9, for parts of the `AIRLINE-INFO` structure of Example 1, the sets are implemented as a linked list. Insertions to the linked lists are done at the end of the linked list and deletions are performed by flagging. Other variations of the linked list like double pointers and coral rings could be used to improve performance. However, to access a tuple when the hierarchical tid is known is tedious as a sequential traversal through linked lists is required, thus violating our goal of minimizing the disk accesses when the tid is given.

3.3.2. Array Pointer Structure

In the simple pointer scheme it is difficult to access a sub-tuple when we have the hierarchical address because to get to the sub-tuple one has to traverse through all the intermediate pointers. To circumvent this problem all pointers within a tuple are moved to one single block – the structure block, which stores only the structural information as shown in Figure 10. Data pages now contain uninterpreted data [7, 8]. The Array-Pointer structure for parts of the `AIRLINE-INFO` structure is illustrated in Figure 11. In this scheme it is possible to get to any value of the sub-tuple in exactly two disk accesses.

There are several issues that must be considered when using this scheme.

1. If we keep a pointer for each sub-tuple in the structure node it may become too large to fit in one disk block.
2. As the cardinality of a set is not fixed the structure nodes are dynamic. Therefore, it is not possible to predetermine the size of the structure node accurately.

These problems can be handled as follows:

1. Instead of having a pointer for each sub-tuple, several sub-tuples can be grouped together on one page, and instead of storing a separate pointer for each sub-tuple, one pointer is stored for a group of sub-tuples stored contiguously in the data pages. The exact location can be computed by using the base pointer and a displacement computed using the size of the sub-tuple. This reduces the number of pointers in the structure node but forces the data pages to store sub-tuples of the same kind. While this is not necessarily a problem, it is interesting to observe that this approach has a flattening effect on data.
2. It is not unreasonable to ask the DBA to specify typical cardinality for the set components at the time of the definition of the schema. This could be used as a 'guide' when designing the sizes of the pointer arrays. In case the array overflows, a pointer to link to the next pointer array is also stored.

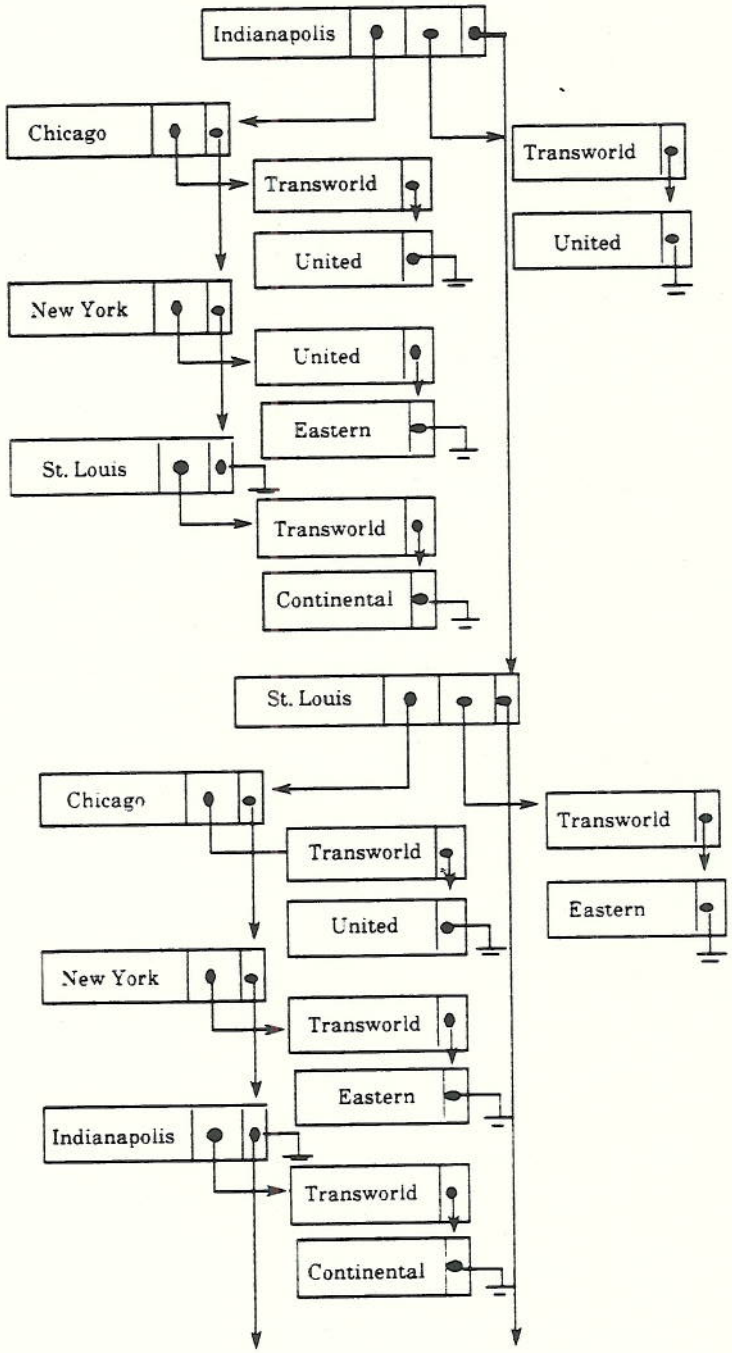

Figure 9: The Simple Pointer Structure

3. In case the structure nodes become too large it is possible to have a hierarchy of structure nodes. This structure makes the file look like an Indexed Sequential File.

Two important advantages of the Array-Pointer scheme are:

1. This structure allows the relocation of data pages without having to alter tuple-

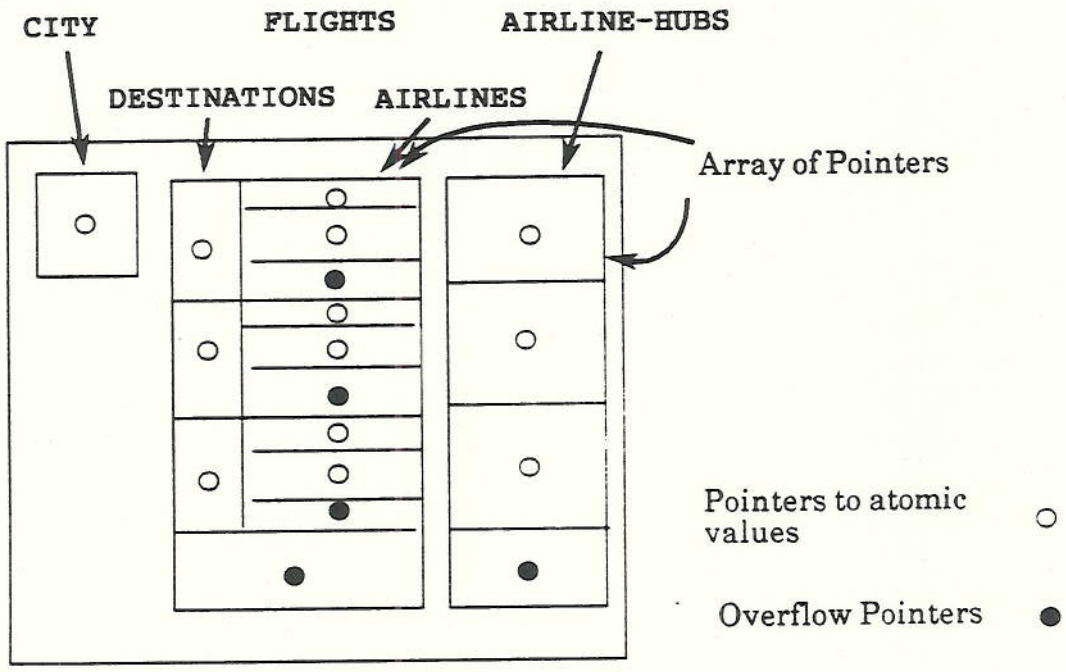

Figure 10: The Structure Node for the Array-Pointer scheme

identifiers. This is very important because tuple-identifiers are also used by the VALTREE and their stability is very critical.

2. Structure nodes for several tuples that would be required frequently could be cached thereby reducing disk accesses to the structure node. The VALTREE provides an indication of which tuples may be required by the query.

3.3.3. Hashing Scheme

The objective of the structure node in the Array-Pointer scheme is to map the hierarchical tuple-identifier to the actual physical address. It is natural to consider a hashing scheme for this problem. Hashing could be done on the key values of the sub-tuple or on the tuple-identifiers of the sub-tuple.

Hashing on key values is not necessary as the VALTREE takes care of value-driven indexing. Since we are also interested in the ability to traverse all tuples and sub-tuples of the structure, hashing on key-values is not an appropriate scheme to do so.

The second alternative is hashing on tuple-identifiers. This scheme needs to be considered in the context of the granularity of the data and search, insert, delete and traverse operators.

Granularity : It is convenient to have uniform buckets so that the slots in the buckets are of the same size. This could be done by storing different kinds of sub-tuples in different hash areas. This is not a serious problem and it is a trade-off between the number

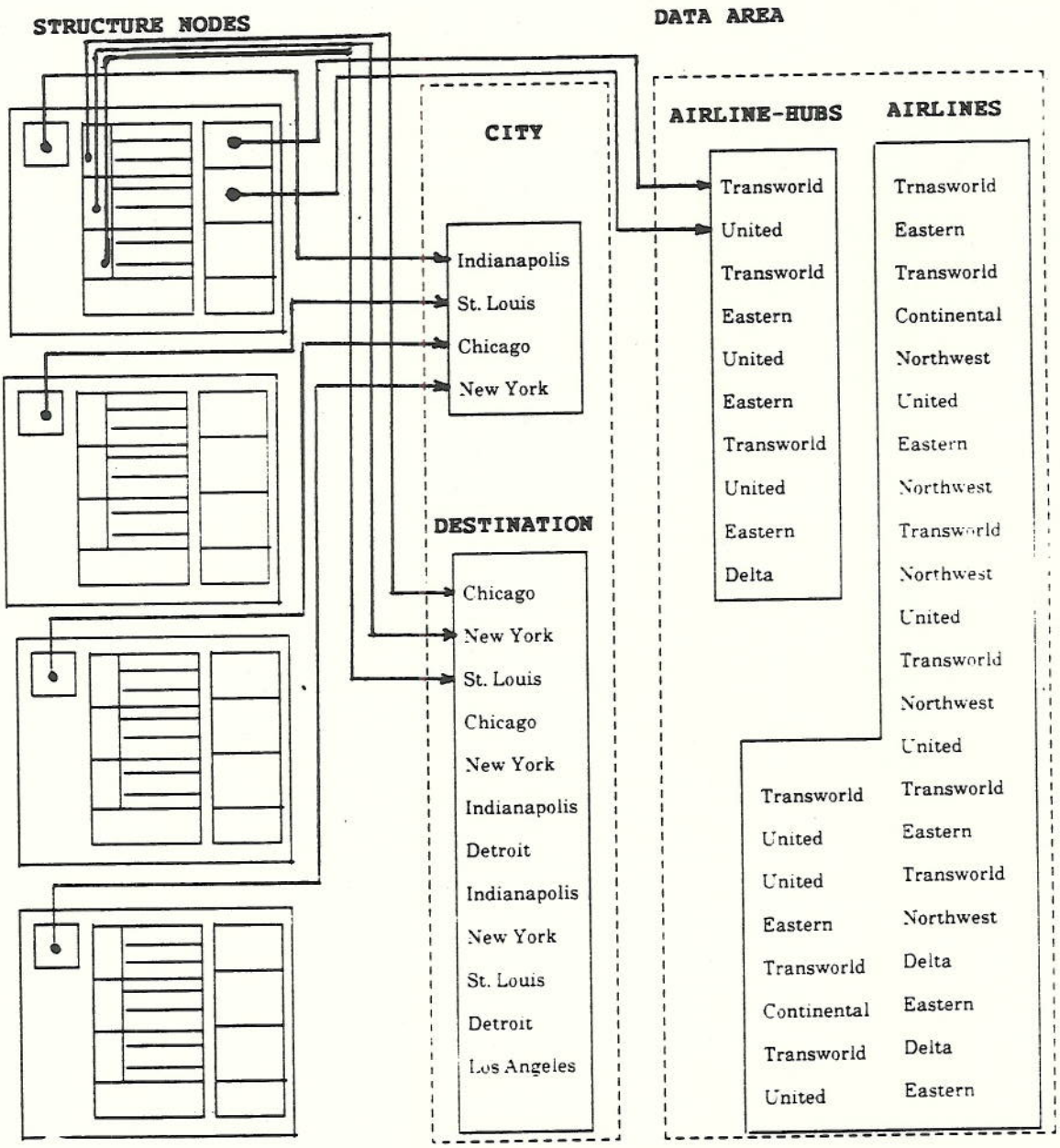

Figure 11: Array Pointer Structure

of disk-accesses required to reconstruct a tuple and the granularity of the operations on parts of the sub-tuple. Thus if we have sub-tuples whose components would seldom be accessed individually then it makes sense to store all components of the sub-tuple in one slot of a bucket. This issue is discussed again in the section about granularity of operations in this model.

Search : To search a sub-tuple given by a tid, say $t_x{}^y{}_z$, we generate a hash-value from the tid which maps us to the bucket that contains the appropriate sub-tuple. As several

tids map to the same bucket, it will be difficult to associate the appropriate sub-tuple to the tid unless that tid is stored along with the corresponding sub-tuple. This is at the cost of extra storage space. It must however be noted that this extra storage space is in the bucket and does not cost any more disk accesses.

Insert : To insert a tuple in the RECLIST structure one has to generate a new hierarchical tid before the tuple can be mapped to the appropriate bucket by the hash function. Hence it is required to save at each tuple and sub-tuple level the last generated tid for that level. Doing this bookkeeping will require some additional space. We propose the following two suggestions to handle this additional space problem:

1. We can have a structure node similar to the Array-Pointer scheme which stores counters instead of pointers for each tuple.
   - This may be better than the previous scheme because counters are not required for search and delete. They would b only be required for insert and for traverse.
   - The structure node is less dynamic than the Array-Pointer scheme since instead of keeping pointers for each sub-tuple all we need to store is the count. It should be noted however, that this structure is not totally static because we need one counter for each set of sub-tuples, in addition the sub-tuple can in turn have many sets of sub-tuples.
   - As the counters that need to be stored are smaller than pointers, the structure node for the Hashing scheme would be smaller than the Array-Pointer scheme. Thus, it should be possible to cache more structure nodes, thus reducing the number of disk accesses.
2. Instead of having all counters stored together in one structure node the counters could be placed along with the data pages. For example, if we number all tuples and sub-tuples starting with 1, the 0 value could be used to map to the slot in the data area which stores a counter instead of data-values. Thus the slot that corresponds to t_0 stores the counter for the number of tuples in the structure t.

Delete : It is not possible to reclaim tids by compression as this will involve updating all the tids of the VALTREE. Thus deletions are performed by flagging. However, it is possible to reclaim tids when inserting new sub-tuples.

Traverse : There are two kinds of traversals possible. Traversal within a tuple, i.e. the reconstruction of the tuple, and traversals across all tuples of the structure. Both kinds of traversals are performed by generating tids in order, until the maximum value is reached and then reconstructing the entire tuple.

Delete flags create holes in the sequence generated for traversal. Each time the tuple-identifier comes across a deleted sub-tuple a disk-access is wasted. A solution to this problem would be to store the delete flags in the structure node instead of storing them with the sub-tuples. This is done by storing a bitmap, where a bit of the bitmap is set if the corresponding sub-tuple is valid – stored and not deleted. Since the counter value can be computed by checking the largest set bit all counters can be replaced by bitmaps. Again both the schemes proposed to store counters could be used to store the bitmaps. An interesting observation can be made here: the layout of a structure node for the

bitmaps is very similar to the Array-Pointer scheme, with a pointer field replaced by a bit to store the status of the corresponding sub-tuples. Hence all the observations made about the structure node in the Array-Pointer scheme applies to the bitmaps associated with the hashing scheme.

Now, traversal is performed by generating tuple-ids for sub-tuples where the delete flag shows the existence of a valid sub-tuple.

Deletions in this scheme can be performed by simply toggling the appropriate bit in the bitmap and not reclaiming the appropriate space immediately. This space could be reclaimed later by a background process or it can be reclaimed when a new tuple is inserted. This is possible because the tuple-identifier will always correspond to the same hash bucket.

To summarize, the linked list method is not really suitable because it is too slow. However, either the Array-Pointer or the the Hashing method would be appropriate. The Hashing method is better than the Array-Pointer method for the following reasons:

1. Search does not need an access to the structure node.
2. The number of disk-accesses required for the Hashing scheme when the structure node is required is exactly the same as the the number of accesses for the Array-Pointer scheme.
3. The structure node in all variations of the the Hashing scheme is smaller than the structure node of the Array-Pointer scheme.

4. Implementation of NRDM Operators

Our goal is to design a complete database management system based on the nested relational database model. To accomplish our goal, we must be able to perform all of the standard data manipulation commands (`INSERT`, `MODIFY` and `DELETE`) and all extended relational algebra operators (`UNION`, `DIFFERENCE`, `SELECT`, `PROJECT`, `JOIN`, `NEST` and `UNNEST`). In this section, we show how we can implement the extended relational algebra operators and the data manipulation operators of the NRDM in terms of our storage scheme. We illustrate by using Example 1, how our storage scheme is put to use in the implementation of the operations of the NRDM.

4.1. Database Maintenance Operators

Insert : Inserts in a tuple can be at the top level – adding a new service from a `CITY` that does not exist in our database, or could be deeply nested – adding a new flight from Indianapolis to Chicago. To ensure that the hierarchical property of nested relations is not compromised, it is important to determine where the value to be inserted exists in the database, an operation appropriate for the `VALTREE`. These inserts are handled as follows:

Example 8: Let us add a United flight from Indianapolis to Minneapolis.

In the AIRLINE-INFO structure, the CITY attribute is the key attribute, hence we first check if 'Indianapolis' is already in the database. This check can be done by looking up the value 'Indianapolis' in the VALTREE data structure. If the value 'Indianapolis' is found, we add the flight to the existing set of FLIGHTS; otherwise we add both 'Indianapolis' and the flight information to the storage structure.

Some important observations:

1. When performing insertions it is important to ensure that all the integrity constraints are satisfied. As will be discussed in the following section the VALTREE is well suited for constraint satisfaction.
2. At every level of the hierarchical structure to be inserted, atomic values must be inserted before set components because atomic values form the key, and all sub-tuples derive their tuple-identifiers from their immediate super-tuple. All the set-valued components at a level may be inserted in any order.

Delete : The strategy for deletion is very similar to that for insertion. To delete a tuple from the database, we need to find the location of the tuple in the RECLIST structure. We find the location for the tuple by looking up the key-value in the VALTREE. We delete a value from the VALTREE if the value to be deleted has only one element in the tuple-identifier list.

Example 9: Discontinue the United service from Chicago to New York.

This deletion can be performed in the following two ways:

1. Determine if the value 'Chicago' exists in the database as a CITY, VALTREE could provide this information easily. Then for each sub-tuple in the FLIGHTS component determine if 'New York' is one of them, RECLIST would be required for this operation. Again traverse the AIRLINES component in the RECLIST and determine if 'United' is one of them. Now the RECLIST is updated by setting deletion flag for the value 'United' and removing the tid from the IDENTIFIER level of VALTREE.
2. Search in the VALTREE and extract all tids that correspond to the appropriate attribute for all the three values 'Chicago', 'New York' and 'United'. The intersection of the three tid sets results in a set of tids which corresponds to the sub-tuples that must be deleted. This is now achieved by actually performing a delete in RECLIST and the VALTREE.

It is interesting to compare the two schemes. The second one is better if the set of identifiers generated by 'Chicago', 'New York' and 'United' is small as compared to the number of sub-tuples that are contained in the AIRLINES component of the sub-tuple with 'New York' as the DESTINATION and the number of sub-tuples contained in the FLIGHTS component with 'Chicago' as the CITY value.

4.2. Data Manipulation Operators

Select : Selects in the NRDM are not restricted only to atomic values but could involve sets as well.

Example 10: Find cities and Transworld flights originating at Transworld hubs.

Observe that this query is value-driven. The VALTREE data-structure is particularly appropriate in this situation. When we search for 'Transworld' at the VALUE level, we find AIRLINES and AIRLINE-HUB attributes at the ATTRIBUTE level if there are Transworld flights arriving at cities with 'Transworld' hubs. The intersection of the lists of tuple-identifiers under these attributes will be non-empty and will correspond to list of desired tuple-identifiers.

Join : Joins are anticipated to occur less frequently in the NRDM since nested relations store some joins implicitly. According to the algebra defined in Section 2, the join operation is defined only on atomic values. Therefore the VALTREE is well suited for the operation. According to the strategy proposed by Missikoff [17, 18], we traverse through the VALTREE and get lists for each value appearing in both structures under the pivot attributes. We then combine tuples identified by the two lists by extracting the tuples from the RECLIST. In Figure 6, we perform a join between SCHEDULE'' and the FLIGHTS' structure and the pivot attributes for this join are NEAREST-AIRPORT for the SCHEDULE'' structure and DESTINATION for the FLIGHTS' structure. As the domains of the pivot attributes are compatible the values for both the attributes will be stored together in the VALUE level of the VALTREE. To perform a join, we traverse through the VALUE level for the domain for the two attributes. If the ATTRIBUTE and the STRUCTURE levels have identifier sets for both the pivot attributes then the value will participate in the join, else the value will not participate in the join.

There are two advantages of this approach for performing joins:

1. Join is performed by traversing all values of the VALUE level of the VALTREE once. Thus the complexity of join is linear with respect the number of values at this level.
2. The hierarchical structure of the joined relation is maintained as the VALTREE ensures that all values are stored only once.

Project : A simple project operation is a structure-oriented operation and hence we do not need to use the VALTREE. Instead, we use the RECLIST. The tuple-identifiers for components to be projected can generated fairly easily. Thus the tids for the projection shown in Figure 6 are

$$t_x{}^b, \qquad where \quad 1 \leq x \leq (\text{maximum number of tuples})$$

These tuple-ids can now be hashed to appropriate buckets to extract complete sub-tuples.

However, the extended project operator as discussed in Section 2, maintains the hierarchical structure. The extended project is performed by traversing the VALTREE for the key attribute and constructing the required parts of the tuple by generating appropriate tids from the set of tids that corresponds to the key value.

Nest :

Example 11: : Let us consider the nest example as shown in Figure 4.

To perform the nest operation, we go through the VALTREE and extract all the identifiers corresponding to each of the values under the NEAREST-AIRPORT attribute. For each value, under the NEAREST-AIRPORT attribute and the SCHEDULE' structure we get a set of tids which correspond to all occurrences of that value. For each element of the set of tids, we reconstruct the tuples by extracting the rest of the values from the RECLIST structure.

Unnest : This is a structure oriented operation. Unnests can be performed very easily by traversing through the entire RECLIST. This is done by generating all the tids as shown for projects and then reconstructing the tuples, repeating values where necessary.

5. Discussion

In this section, we discuss how our storage structure is suitable to effectively handle some other important DBMS issues.

5.1. The VALTREE as a Nested Relational Structure

The VALTREE itself can be thought of as a nested relation as shown in Figure 12. This allows one to perform nested relational algebraic operations on the VALTREE. This allows for example to consider other indexing schemes like the standard (attribute, value) pairs by simply restructuring the VALTREE using the nested relational algebra.

If fast implementations for the RECLIST structure become available, the VALTREE can be implemented as a RECLIST and all the VALTREE operations can be performed by performing algebraic operations on VALTREE stored as a RECLIST.

5.2. Object-Oriented Databases

Object-Oriented Databases are becoming increasingly popular. Our research would be beneficial to the implementation of object-oriented databases in the following two ways:

1. Several current implementations of object-oriented databases map the object oriented systems to relational databases. While this is possible, designers of such systems have problems mapping complex objects to flat relations. We feel that the mapping from object-oriented databases to nested relational databases, though not entirely trivial, is much cleaner than the mapping to a relational model. This is because the nested relational paradigm models sets which are fundamental to object-oriented systems.
2. The problems faced by designers building 'pure' object-oriented systems [10] are very similar to the problems that are faced in the representation of the nested relational model. Data-structures like the VALTREE and the RECLIST with some modifications could be used for designing object-oriented databases. The notation for tuple-identifiers is similar to the tagged notation used for creating object-identifiers.

DOMAIN	VALUE	ATTRIBUTE	STRUCTURE	{IDENTIFIER}
City Name	Atlanta	Destination	Airline-Info	$\{t_4{}^{b}{}_5{}^{a}\}$
	Chicago	City	Airline-Info	$\{t_3{}^{a}\}$
		Destination	Airline-Info	$\{t_1{}^{b}{}_1{}^{a}, t_2{}^{b}{}_1{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_2{}^{b}{}_2{}^{a}, s_3{}^{b}{}_2{}^{a}, s_3{}^{b}{}_1{}^{a}, s_6{}^{b}{}_1{}^{a}\}$
	Cincinnati	Destination	Airline-Info	$\{t_4{}^{b}{}_4{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_1{}^{b}{}_2{}^{a}\}$
	Detroit	Destination	Airline-Info	$\{t_2{}^{b}{}_4{}^{a}, t_3{}^{b}{}_4{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_4{}^{b}{}_1{}^{a}, s_5{}^{b}{}_1{}^{a}\}$
	Indianapolis	City	Airline-Info	$\{t_1{}^{a}\}$
		Destination	Airline-Info	$\{t_2{}^{b}{}_3{}^{a}, t_3{}^{b}{}_1{}^{a}, t_4{}^{b}{}_1{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_1{}^{b}{}_1{}^{a}, s_2{}^{b}{}_1{}^{a}\}$
	Los Angeles	Destination	Airline-Info	$\{t_3{}^{b}{}_5{}^{a}\}$
	Louisville	Nearest-Airports	Schedule	$\{s_1{}^{b}{}_3{}^{a}\}$
	New York	City	Airline-Info	$\{t_4{}^{a}\}$
		Destination	Airline-Info	$\{t_1{}^{b}{}_2{}^{a}, t_2{}^{b}{}_2{}^{a}, t_3{}^{b}{}_2{}^{a}\}$
	St. Louis	City	Airline-Info	$\{t_2{}^{a}\}$
		Destination	Airline-Info	$\{t_1{}^{b}{}_3{}^{a}, t_3{}^{b}{}_3{}^{a}, t_4{}^{b}{}_2{}^{a}\}$
		Nearest-Airports	Schedule	$\{s_6{}^{b}{}_2{}^{a}\}$
Airline Name	...	...	...	{...}
Team Name	...	...	...	{...}

Figure 12: The VALTREE as a nested relational structure

Some more interesting solutions for problems with object-oriented systems, such as object sharing, can be handled by associating object-ids explicitly with objects and storing these object-identifiers in the VALTREE as though they were the values.

5.3. Granularity of the Database

While it may be ideal to save every atomic value in the VALTREE and have a pointer for each atomic value in the structure node of the RECLIST, this may not be appropriate or feasible. It is therefore left to the DBA to adjust the granularity of indexing. Thus tuples which are always accessed together and never as components may be stored as a single entity in the RECLIST and the key value for the tuple may be stored in the

VALTREE instead of storing all individual values.

5.4. Integrity Checking

When we perform an insert or delete we have to check if all the integrity constraints have been satisfied. This checking is based on a value-driven approach. For instance, when we say that $A \rightarrow B$, we mean that when the values of attribute A match in two tuples they must match in values of B. This can be checked by looking for each value corresponding to attribute A in the VALTREE. We get a set of tids, say S_A. We pick any one tid from the set S_A and then using the RECLIST find a corresponding value, v_B for attribute B. Now we can go to the VALTREE and extract the set of tids, S_B that correspond to the value v_B and attribute B. The integrity constraint is satisfied if the set of identifiers associated with the B-value, S_B is a super-set of the list of tuple-identifiers of the A-value S_A. Other constraints also require that values for certain attribute obey a set of criteria. Conditions can therefore be verified while performing inserts in the VALTREE.

5.5. Intermediate Results

Most database queries are performed in stages, thus intermediate results are very important. As our algorithms depend on the use of two data structures it may be important to maintain the two data-structures for all partial results. We have not yet studied the issue of intermediate results in detail. Several approaches to this problem could include:

1. Do not maintain any new data-structures on partial results; use tids and extract from the same VALTREE and RECLIST all the values as and when needed.
2. Assume that the partial result is a new structure and store the structure as a RECLIST and add values to the existing VALTREE.
3. Generate new, small and temporary VALTREE and RECLIST structures which survive only until the expression has been evaluated.

5.6. Query Optimization

This is another issue that has not been studied in detail for nested relational models. The VALTREE and the RECLIST are an integral part of our storage scheme and they should be exploited to perform query optimization. Furthermore, while the algebraic properties for the nested algebra are fairly well understood, as was demonstrated in some of the examples of the previous section, alternate query plans for the same query are possible. We believe that query optimization should not only take into account the algebraic properties but should also consider heuristics and the current state of the database. We are currently involved in studying this problem. While it is possible to draw parallels from the query optimization techniques for the relational model, these techniques cannot be mapped directly to the NRDM as additional problems need to be addressed.

5.7. Partitioning and Parallelism

Nested structures inherently partition the data horizontally. Another level of partitioning of the data occurs in the VALTREE. For instance, the tuples in a structure are partitioned according to the values they have. We can effectively set up locks at each value level thereby allowing us to use concurrent processes to perform our operations. When we are performing an update, we need to lock only the concerned values and do not need to lock the entire database. This approach lets us localize in memory our most active and interacting processes. Furthermore, partitioning of the database allows us to perform several operations in parallel.

5.8. Computing Transitive Closures

Example 12: Let us consider the standard manager subordinate example. Let us say we pick the chairman who has some subordinates. Each of the subordinates in turn have some subordinates and so on. Assume we want to find all the subordinates of the chairman. To handle this example in our scheme, we first select the chairman. Now as the chairman is the manager of some subordinates, his name must appear as the manager attribute of those tuples. We get the tuple identifiers for all the subordinates as soon as we search for the chairman in the value-driven tree. Once we have the tuple identifiers for each of the subordinate tuples we can extract the names of the subordinates from the RECLIST structure. Once we have the names of the subordinates at the first level, we can, possibly in parallel, search all their sub-ordinates in a similar manner.

Example 13: : Determine all cities which have direct or indirect flights to St. Louis.

This problem can be solved as follows:

1. Since we are interested in all cities connected to city of 'St. Louis', we must find all cities that have flights with 'St. Louis' as the destination. To do this we look up the VALTREE and collect all tids where 'St. Louis' is the destination.
2. Now, for each tid obtained from the previous step we extract the tid for the CITY component and extract the city name from the RECLIST.

Now for each of the cities obtained from step 2 we repeat step 1 and 2, alternating selections between RECLIST and VALTREE. If the relationship between these sets is an arbitrary graph then it is possible to repeat indefinitely. To avoid this problem we need to keep a list of all cities included and include a new city only if it is not already there. The algorithm stops when no more cities can be generated. Notice that it is convenient to store a list of tids corresponding to the city component rather than storing city names.

5.9. Logic Programming Interface

There are numerous advantages to coupling logic programming with relational databases[13]. Beeri et.al. have proposed a logic language with sets which could be mapped to the NRDM[5]. Most deductive systems are based on resolution and unification principles which are value-driven in nature. We believe that our storage structure would be appropriate for supporting such deductive systems.

6. References

1. S. Abiteboul, N. Bidoit, "Non First Normal Form Relations to Represent Hierarchically Organized Data", Proc. Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1984, 191-200.
2. F. Bancilhon, "A Logic-Programming/Object-Oriented Cocktail", SIGMOD Record, Vol. 15, No. 3 (Sept. 1986), pp. 11-20.
3. F. Bancilhon, S. Khoshafian, "A Calculus for Complex Objects" Proc. Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1986, 53-59.
4. F. Bancilhon, P. Richard, M. Scholl, "On Line Processing of Compacted Relations", *Proc.8th VLDB*, 1982, 263-269.
5. C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, S. Tsur, "Sets and Negation in Logic Database Language (LDL1)" *Proc. 6th PODS*, San Diego, 1987, pp. 21-37.
6. N. Bidoit, "Efficient Evaluation of Relational Queries Using Nested Realtions", *Rapports de Recherche, no 480*, INRIA, 1986
7. P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, G. Walch, "A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Washington, D.C., 1986, 356-366.
8. U. Deppisch, H.-B. Paul, H.-J. Schek, "A Storage System for Complex Objects", Proc. of the Int'l Workshop on Object-Oriented Database System, Pacific Grove, 1986, pp. 183-195.
9. A. Deshpande, D. Van Gucht, "A Storage Structure for Unnormalized Realtions", *Proc. GI Conf. on Database Systems for Office Automation, Engineering and Scientific Applications*, Darmstadt, April 1987, pp. 481-486.
10. U.Dayal et.al. "The Probe Data Model" *Proc. GI Conf. on Database Systems for Office Automation, Engineering and Scientific Applications*, Darmstadt, April 1987.
11. G. Houben, J. Paredaens, "The R^2-Algebra: An Extension of an Algebra for Nested Relations", *Tech. Rep.*, Tech. University, Eindhoven, 1987
12. G. Jaeschke, H.-J. Schek, "Remarks on the the Algebra on Non First Normal Form Relations", Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1982, 124-138.
13. L. Kerschberg ed., *Expert Database Systems - Proceedings from the First International Workshop*, Benjamin/Cummings Publishing Company, Inc. 1986.
14. G.M. Kuper, "Logic Programming With Sets", *Proc. 6th PODS*, San Diego, 1987, pp. 11-20
15. V. Linnemann, "Non First Normal Form Relations and Recursive Queries: An SQL-Based Approach", *Proc. 3rd IEEE Int. Conf. on Data Engineering*, Los Angeles, 1987
16. A. Makinouchi, "A Consideration of Normal Form of Not-Necessarily-Normalized Relations in the Relational Data Model", Proc. 5th Int'l Conf. on Very Large Data Bases, 1977, 447-453.

17. M. Missikoff, "A Domain Based Internal Schema for Relational Database Machines", Proc. ACM SIGMOD Int'l Conf. on Management of Data, 1982, 215-224.

18. M. Missikoff and M. Scholl, "Relational Queries in Domain Based DBMS", Proc. ACM SIGMOD Int'l Conf. on Management of Data, 1983, 219-227.

19. H.-B. Paul, H.-J. Schek, M.H. Scholl, G. Weikum, U. Deppisch, "Architecture and Implementation of Darmstadt Database Kernel System" *Proc. Ann SIGMOD Conf.*, San Fransisco, 1987, pp. 196-207.

20. H.-B. Paul, A. Söder, H.-J. Schek, G. Weikum, "Unterstützung der Büro-Ablage-Service durch ein Datenbankkernsystem" GI-Fachtagung Datenbanksysteme in Büro, Technik und Wissenschaft, Darmstadt, 1987, pp. 198-211

21. P. Pistor, F. Andersen, "Designing a Generalizied NF^2 Model with an SQL-Type Language Interface", *Proc. 12th VLDB*, Kyoto, Japan, 1986, pp. 278-288.

22. P. Pistor, R. Traunmueller, "A Database Language for Sets, Lists and Tables", *Information Systems 11:4*, 1986, pp. 323-336

23. M.A. Roth, H.F. Korth, D.S. Batory, "SQL/NF: A Query Language for ¬1NF Relational Databases", Tech. Report TR-84-36, University of Texas at Austin, 1984.

24. M.A. Roth, H.F. Korth, A. Silberschatz, "Theory of Non-First-Normal-Form Relational Databases", *Tech. Report TR-84-36 (Revised January 1986)*, University of Texas at Austin, 1984.

25. M.H. Scholl, H.-B. Paul, H.-J. Schek "Supporting Flat Relations by a Nested Relational Kernel" *Proc. 13th VLDB*, London, 87

26. S.J. Thomas, P.C. Fischer, "Nested Relational Structures", *Advances in Computing Research III, The Theory of Databases*, P.C. Kanellakis, ed., JAI Press, 1986, pp. 269 - 307.

27. D. Van Gucht, P.C. Fischer, "High Level Data Manipulation Languages for Unnormalized Relational Database Models", Tech. Report, Indiana University, 1986.